International Standard

ELF 5002

EXPRESS language pretty print format

First edition
2025-05

Reference number
ELF 5002:2025(en)

COPYRIGHT PROTECTED DOCUMENT

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword

The EXPRESS Language Foundation (“ELF”) is a registered public charity in the US that facilitates the education, standardization, research, promotion, definition, and usage of information modelling and programming languages, with a focus on the EXPRESS language family.

ELF works with international partners and experts across the globe, reflecting the international nature of its mission. More information about ELF is available on the official website (https://www.expresslang.org).

The procedures used to develop this document and those intended for its further maintenance are described in the ELF Directives.

In particular, the different approval criteria needed for the different types of ELF documents should be noted. This document was drafted in accordance with the editorial rules of the ELF Directives.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ELF shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be provided in the Introduction.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

This document was prepared by Technical Committee *EXPRESS*.

Introduction

The EXPRESS language, defined in ISO 10303-11, is widely used for data modeling in product data exchange applications. A consistent, readable formatting of EXPRESS schemas is crucial for human readers to understand the structure and semantics of the data models.

Pretty printing refers to the systematic formatting of source code, in this case EXPRESS schemas, to enhance readability, maintainability, and consistency. A standardized pretty printing format ensures that all EXPRESS schemas follow uniform layout rules, making them easier to read, understand, and maintain across different implementations and applications.

The rules specified in this document were originally developed by the Express Engine team to fill a gap in the ISO/TC 184/SC 4 Supplementary Directives. While the Supplementary Directives document (TC 184/SC 4 N2412) and its subsequent versions provide some guidance on EXPRESS schema formatting, they do not comprehensively address the layout of all EXPRESS constructs.

The pretty printing rules presented here explain how various EXPRESS constructs should be formatted for presentation. The document provides references for the formatting choices where possible, especially when they are derived from the SC4 Supplementary Directives. Where the Directives do not specify formatting rules, this document establishes consistent formatting patterns based on the Express Engine implementation.

This standard addresses the layout of three basic groups of objects in EXPRESS:

— Declarations (schema, entity, type, function, etc.)

— Expressions (arithmetic, logical, relational, etc.)

— Statements (assignment, procedure calls, conditionals, etc.)

By following these rules, EXPRESS schema authors and tool developers can ensure that their schemas are presented in a consistent, readable manner that aligns with community practices.

This standard builds upon the significant contributions of the Express Engine team, particularly the work of Craig Lanning and Thomas Thurman who authored the original documentation of pretty printing rules for the EXPRESS language. The EXPRESS community recognizes their valuable contributions to establishing consistent and readable formatting for EXPRESS schemas.

While the ISO/TC 184/SC 4 had not previously defined comprehensive pretty printing rules for EXPRESS, the Express Engine team contributed these rules to the ISO/TC 184/SC 4 Supplemental Directives. The EXPRESS Language Foundation has adopted these rules as a formal standard to ensure consistent schema formatting across the EXPRESS ecosystem.

EXPRESS language pretty print format

1 Scope

This document specifies rules for the presentation format (pretty printing) of EXPRESS language schemas. It provides a standardized approach to formatting various elements of the EXPRESS language, including declarations, expressions, and statements.

The rules specified in this document apply to all EXPRESS schemas, regardless of their application domain. The standardized pretty printing format enhances readability, improves maintainability, and ensures consistency across EXPRESS implementations.

This document provides test cases and examples to illustrate the formatting rules and their application to different EXPRESS constructs, and can be used by a EXPRESS pretty printer to claim conformance to this standard.

This document is intended for:

— EXPRESS schema authors who wish to format their schemas in a consistent, readable manner

— Tool developers implementing pretty printing capabilities for EXPRESS schemas

— Reviewers and users of EXPRESS schemas who benefit from consistently formatted schemas

The pretty printing rules specified here complement the syntax and semantics of the EXPRESS language as defined in ISO 10303-11.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 10303-11:2004, *Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual*

ISO/TC 184/SC 4 N2412, *SC 4 Supplementary directives — Rules for the structure and drafting of SC 4 standards for industrial data*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org

3.1
pretty printing
systematic formatting of source code according to predefined rules to enhance readability, maintainability, and consistency

3.2
pretty printer
software tool or library that applies pretty printing rules to source code, transforming it into a more readable and consistent format

3.3
declaration
EXPRESS language construct that defines schemas, entities, types, functions, rules, and other structural elements of a data model

[SOURCE: ISO 10303-11:2004]

3.4
expression
combination of operands and operators that computes a value

[SOURCE: ISO 10303-11:2004]

3.5
statement
executable instruction in the EXPRESS language that performs an action such as assignment, procedure call, or control flow operations

[SOURCE: ISO 10303-11:2004]

3.6
SC4 Supplementary Directives
document published by ISO/TC 184/SC 4 that provides additional guidance for the development and presentation of standards, including formatting rules for EXPRESS schemas

[SOURCE: ISO/TC 184/SC 4 N2412]

4 Principles

4.1 Pretty printing

Pretty printing is the systematic formatting of source code according to predefined rules to enhance readability, maintainability, and consistency.

The practice of pretty printing is common in programming languages and software development. It involves organizing code into a structured format that is easy to read and understand. This includes consistent indentation, line breaks, and spacing, as well as the use of meaningful names for variables, functions, and other constructs. Pretty printing also helps identify syntax errors and improve the overall quality of the code.

Pretty printing is particularly important in the context of data modeling and schema definition languages like EXPRESS. In these languages, the structure and semantics of the data models are defined through a series of declarations, expressions, and statements. A well-formatted schema makes it easier for developers, data architects, and other stakeholders to understand the relationships between different entities, types, and attributes. It also aids in the maintenance and evolution of the schema over time, as changes can be made more easily when the code is organized and readable.

4.2 EXPRESS pretty printing

In the context of the EXPRESS language, pretty printing refers to the formatting of EXPRESS schemas, entities, types, functions, rules, and other declarations to ensure that they are presented in a consistent and readable manner. This is particularly important for human readers who need to understand the structure and semantics of the data models defined in EXPRESS.

This document is developed based on EXPRESS formatting rules described in the following references:

— the ISO/TC 184/SC 4 Supplementary Directives TC 184/SC 4 N2412 and its later versions

— the Express Engine development guide for its Pretty Printer, where it covers formatting rules beyond the scope of the SC4 Supplementary Directives

NOTE The rules specified in this document were originally developed by the Express Engine team to fill a gap in the ISO/TC 184/SC 4 Supplementary Directives. While the Supplementary Directives document (TC 184/SC 4 N2412) and its subsequent versions provide some guidance on EXPRESS schema formatting, they do not comprehensively address the layout of all EXPRESS constructs.

EXPRESS pretty printing involves the formatting of its various constructs, covered by the three basic groups of objects:

— Declarations

— Expressions

— Statements

5 EXPRESS pretty printing parameters

5.1 General

In a pretty printing process, parameters are used to control the formatting of the output. These parameters can include settings for indentation, line length, and other formatting options. The parameters are typically specified in a configuration file or as command-line arguments when invoking the pretty printer.

5.2 Pretty printing parameters

This following parameters are used to control the formatting of the output in the context of EXPRESS pretty printing.

`indent` Indent specifies the number of spaces to use for indentation. This parameter can be set to a specific value, such as 2, 4, or 8, or to a special value like `tab` to indicate that tabs should be used for indentation. The default value is 4 spaces.

`line-length` Line length specifies the maximum number of characters to allow on a single line before wrapping to the next line. This parameter can be set to a specific value, such as 80 or 120 or to a special value like `unlimited` to indicate that there is no limit on line length.

`insert-newline` Insert newline specifies whether to insert a newline character after certain constructs, such as declarations or expressions. This parameter can be set to `true` or `false`, depending on the desired formatting style.

`insert-space` Insert space specifies whether to insert a space character between certain constructs, such as operators or parentheses. This parameter can also be set to `true` or `false`, depending on the desired formatting style.

6 Declarations

6.1 General

This clause describes the layout of each declaration: FILE header, SCHEMA, interface, CONSTANT, TYPE, ENTITY, SUBTYPE_CONSTRAINT, FUNCTION, RULE, and PROCEDURE.

6.2 Preamble

The preamble is printed as follows:

a) Remark items, delimited by the open and close remark symbols ((* and *)) that form the initial lines in the input file, starting with the first line and ending with the line immediately before the line with the SCHEMA keyword, are printed individiually separated by new lines, flush to the left.

b) One blank line is printed after the last remark item, if any.

c) Provenance information is printed as a remark item, starting with the (* symbol and ending with the *) symbol. Provenance information includes the name of the pretty printer, the version of the pretty printer, and the parameters used to execute the pretty printer.

NOTE This clause is intended to comply with SC4 and ISO agreements for managing headers for EXPRESS schemas that are publicly available, as well as provide provenance information for downstream applications.

EXAMPLE — Pretty printing of the preamble

```
(* This is a comment 1 *)
(* This is a multi-line comment
     2 *)
(*
This is a multi-line comment 3

*)

(* {pretty-printer} version: {version}
   Format parameters: {format-parameters}
*)
```

6.3 SCHEMA

NOTE This clause reflects ISO/TC 184/SC 4 N2412, 6.1.6, “Layout of schema”.

The SCHEMA element is printed as follows:

a) The SCHEMA and END_SCHEMA keywords shall be flush with the left margin.

b) The SCHEMA keyword shall be on a line immediately after the preamble.

c) Contained objects shall also have the begin and end keywords flush with the left margin.

d) Interface clauses begin flush with the left margin.

EXAMPLE

```
SCHEMA schema_name;

USE FROM schema2;

TYPE type1;
END_TYPE;

ENTITY entity1;
END_ENTITY;

END_SCHEMA;
```

6.4 Interfaces

NOTE This clause reflects ISO/TC 184/SC 4 N2412, 6.1.7, “Layout of interface statements”.

`USE FROM` or `REFERENCE FROM` statements are used to specify the interface between schemas.

Interfaces are printed as follows:

a) An interface begins with the `USE FROM` or `REFERENCE FROM` keywords at the left margin. Followed by the name of the SCHEMA being interfaced.

b) Optionally, there can be a list of object names that are interfaced from the specified schema. If this list is present, it will begin on the following line with the open parenthesis positioned two characters to the right of the left margin. Each item in the list is written to a line by itself with the first one being written just after the open parenthesis. The items are separated by a comma. Each subsequent item will be aligned with the one above it. The final item will have the close parenthesis following immediately after it.

c) Finally, the interface specification is terminated by a semicolon.

d) The source document for the interface shall be provided as a trailing comment immediately after the name of the interfaced schema. If the version of the schema is available in the source document, it shall be provided in the output.

EXAMPLE

```
SCHEMA schema3 '{iso standard 10303 part(nn) version(yy) object(1)
geometric_model_schema(3)}';
USE FROM schema1  -- ISO 10303-42 schema1 version 10
  (entity1,
   entity2,
   entity3);
USE FROM schema2  -- ISO 10303-41 schema2 version 16
  (entity1,
   entity2,
   entity3,
   entity4);
USE FROM schema3; -- ISO 10303-43 schema3 version 5
...
END_SCHEMA; -- schema3
```

6.5 CONSTANT

NOTE This clause reflects ISO/TC 184/SC 4 N2412, 6.1.8, “Layout of constant declaration”.

A constant declaration is printed as follows:

a) The `CONSTANT` keyword is on a line by itself and flush with the left margin of the enclosing declaration.

b) On each following line is a constant specification. Indented two characters to the right of the start of the `CONSTANT` keyword is the name of the constant followed by a colon (:) followed by the declared type of the constant followed by an assignment operator (:=) followed by an expression which calculates the value for the constant followed by a semicolon (;) to terminate the constant definition.

c) All of the colons separating the constant name from the constant type should be aligned. There should be at least one space before and after this colon.

d) All of the colons that are part of the assignment operator which separates the constant type from the initialization expression should be aligned. There should be at least one space before and after the assignment operator.

EXAMPLE

```
CONSTANT
  schema_prefix         : STRING       := 'MATH_FUNCTIONS_SCHEMA.';
  the_integers          : e_space      := mk_e_space(es_integers);
  the_reals             : e_space      := mk_e_space(es_reals);
  the_numbers           : e_space      := mk_e_space(es_numbers);
  the_logicals          : e_space      := mk_e_space(es_logicals);
  the_booleans          : e_space      := mk_e_space(es_booleans);
  the_strings           : e_space      := mk_e_space(es_strings);
  the_binarys           : e_space      := mk_e_space(es_binarys);
```

```
  the_maths_spaces      : e_space      := mk_e_space(es_maths_spaces);
  the_generics          : e_space      := mk_e_space(es_generics);
  the_empty_space       : finite_space := mk_finite_space([]);
  the_nonnegative_reals : real_interval_from_min
                                       := make_real_interval_from_min(0.0, closed);
  the_zero_one_interval : finite_real_interval
                                       := make_finite_real_interval(0.0, closed, 1.0,
closed);
  the_zero_pi_interval  : finite_real_interval
                                       := make_finite_real_interval(0.0, closed, pi,
closed);
  the_neg1_one_interval : finite_real_interval
                                       := make_finite_real_interval(-1.0, closed,
1.0, closed);
END_CONSTANT;
```

6.6 TYPE

NOTE 1 This clause reflects ISO/TC 184/SC 4 N2412, 6.1.9, “TYPE Layout”.

A type declaration is printed as follows:

a) The `TYPE` and `END_TYPE` keywords are flush with the left margin and each on its own line.

b) Immediately after the `TYPE` keyword is the name of the type, an equal sign (’=’) and the underlying type. If the underlying type is either an ENUMERATION or a SELECT and all of the values will not fit on one line, then the list begins on the next line indented two characters and each value will be written on its own line.

c) If present, the `WHERE` keyword is flush with the left margin. The individual where rules are indented two characters to the right.

NOTE 2 This specification of the WHERE clause is not specified in ISO/TC 184/SC 4 N2412, but is consistent with the specification of WHERE clauses in the ENTITY declaration.

EXAMPLE

```
TYPE type1 = STRING;
WHERE
  WR1: <expression>;
END_TYPE; -- type1

TYPE enum_type1 = ENUMERATION OF (on, off, whatever);
END_TYPE; -- enum_type1

TYPE enum_type2 = ENUMERATION OF
  (val1,
   val2,
   val3,
   val4);
END_TYPE; -- enum_type2

TYPE sel_type1 = SELECT (ent1, ent2, ent3);
END_TYPE; -- sel_type1

TYPE sel_type2 = SELECT
  (entity1,
   entity2,
   entity3,
   type4,
   entity5);
END_TYPE; -- sel_type2
```

6.7 ENTITY

NOTE 1 This clause reflects ISO/TC 184/SC 4 N2412, 6.1.11, “Entity data type declaration layout”.

An entity declaration is printed as follows:

a) The ENTITY and END_ENTITY keywords are flush with the left margin and each on its own line.

b) Immediately after the ENTITY keyword is the name of the entity. If there is a supertype specification it is specified next beginning on the next line, indented two characters to the right of the left margin.

c) If there is a subtype specification, it is specified next beginning on the next line, indented two characters to the right of the left margin.

d) Next is a semicolon to mark the end of the entity header.

e) The attributes are next with each one on a line by itself, indented two characters to the right of the left margin. The colons that separate the attribute name from the attribute type should be aligned with at least one space before and after the colon.

f) If there are any derived attributes, they are written next. The DERIVE keyword is written on a line by itself flush with the left margin. Each attribute is written on a line by itself indented two characters to the right of the left margin. The colons that separate the attribute name from the attribute type should be aligned and have at least one space before and after. The colons in the assignment operator should be aligned with at least one space before and after the assignment operator. Each attribute should end with a semicolon.

g) If there are any inverse attributes, they are written next. The INVERSE keyword is written on a line by itself, flush with the left margin. Each attribute is written on a line by itself, indented two characters to the right of the left margin.

h) An inverse attribute has a name followed by a colon and the type of the attribute. The attribute type has an optional SET or BAG aggregate specification which consists of either the SET or BAG keyword followed by an optional bounds specification, followed by the OF keyword and then the name of an ENTITY, followed by the FOR keyword and the specification of the attribute being inverted. The colons between the attribute name and type should be aligned and have at least one space before and after.

   NOTE 2 The specification of the INVERSE keyword and block is not specifically called out in the Directives, but the specification is consistent with the specification of the DERIVE, UNIQUE, and WHERE keywords and blocks.

i) If there are any unique rules, they are written next. The UNIQUE keyword is written on a line by itself, flush with the left margin. Each unique rule is written on a line by itself, indented two characters to the right of the left margin.

j) If there are any where rules, they are written next. The WHERE keyword is written on a line by itself, flush with the left margin. Each where rule is written on a line by itself, indented two characters to the right of the left margin.

EXAMPLE

```
ENTITY entity1
  SUPERTYPE OF (entity2 ANDOR entity3)
  SUBTYPE OF (entity5);
  attr1 : type1;
  attr2 : type2;
DERIVE
  der1 : type1 := <expression>;
INVERSE
  inv1 : entity1 FOR attr3;
  inv2 : BAG OF entity2 FOR attr4;
  inv3 : BAG [1:4] OF entity3 FOR entity2.attr3;
UNIQUE
  UR1: attr1;
  UR2: attr1, attr2;
WHERE
```

```
  WR1: <expression>;
  WR2: <expression>;
END_ENTITY;
```

6.8 SUBTYPE_CONSTRAINT

A subtype constraint declaration is printed as follows:

a) Write the SUBTYPE_CONSTRAINT keyword followed by a space and the name of the subtype constraint declaration, followed by the FOR keyword, a space, the name of the ENTITY that this declaration applies to, and a semicolon.

NOTE The layout of SUBTYPE_CONSTRAINT is not specified in SC4 Supplementary Directives.

a) If this is an abstract declaration then on a new line, indented two characters to the right of the left margin, write the ABSTRACT keyword followed by theSUPERTYPE keyword, followed by a semicolon.

b) If there is a SUPERTYPE OF specification, it is written next, starting on a new line and indented two characters to the right of the left margin. After it is finished a semicolon is written.

c) If there is a total over then on a new line, indented two characters to the right of the left margin, write the TOTAL_OVER keyword followed by an open parenthesis which marks the beginning of the list of entities to total over.

d) If the list of entities will fit on one line then it is done that way, otherwise, each entity is placed on its own line aligned with the one above it. The first entity is written just after the open parenthesis with no space between it and the parenthesis. Between each entity is a comma and a space after the comma. A close parenthesis follows immediately after the last entity. Finally, a semicolon end the total over clause.

e) On a new line, flush with the left margin, write the END_SUBTYPE_CONSTRAINT keyword, followed by a semicolon.

EXAMPLE

```
SUBTYPE_CONSTRAINT entity1_sub_con FOR entity1;
  ABSTRACT SUPERTYPE;
  TOTAL_OVER (entity3, entity4, entity5);
  ONEOF(entity3, entity4, entity5);
END_SUBTYPE_CONSTRAINT;
```

6.9 FUNCTION

NOTE This clause reflects ISO/TC 184/SC 4 N2412, 6.1.10, “Algorithm layout”.

A function declaration is printed as follows:

a) The FUNCTION and END_FUNCTION keywords are on separate lines and aligned with the left margin. Immediately, following the FUNCTION keyword is a space followed by the name of the function.

b) Immediately after the function name, without a space, is the open parenthesis that marks the beginning of the formal parameters. Each parameter is written on a line by itself with the first one being written on the same line as the open parenthesis with no space between it and the parenthesis.

c) A semicolon separates each of the formal parameters and is flush against the parameter that it comes after.

d) After the final formal parameter, is a close parenthesis followed by a colon and then the type for the value that is returned from the function. The colon should have at least one space before and after it. After the return type is a semicolon which marks the end of the function header.

e) If there are any local ENTITY, SUBTYPE_CONSTRAINT, FUNCTION, RULE, PROCEDURE, or TYPE declarations, they are written next, indented two characters to the right of the left margin.

f) If there are any CONSTANT declarations, they are written next. The CONSTANT and END_CONSTANT keywords are written on separate lines, flush with the left margin. Each constant is written on a separate line indented two characters to the right of the left margin.

g) If there are any LOCAL declarations, they are written next. The LOCAL andEND_LOCAL keywords are written on separate lines, flush with the left margin. Each local is written on a separate line indented two characters to the right of the left margin.

h) Finally, we write any statements that are included in the FUNCTION, indented two characters to the right of the left margin.

EXAMPLE

```
FUNCTION fun_name (a : INTEGER;
                   b : STRING;
                   c : REAL) : STRING;
  ENTITY fun_sub_ent;
    name : STRING;
  END_ENTITY;

  CONSTANT
    ...
  END_CONSTANT;

  LOCAL
    inst : fun_sub_ent;
  END_LOCAL;
  inst := fun_sub_ent(b);
END_FUNCTION;
```

6.10 RULE

NOTE This clause reflects ISO/TC 184/SC 4 N2412, 6.1.10, “Algorithm layout”.

a) It starts with the RULE keyword written on a new line flush with the left margin. Immediately following the RULE keyword is a space followed by the name of the rule, followed by the FOR keyword.

b) Immediately after the FOR keyword is an open parenthesis that marks the beginning of the list of entities that this rule applies to. Each entity name is placed on a line by itself with the first one being written on the same line as the open parenthesis with no space between it and the parenthesis.

c) A comma separates each entity name and is positioned flush against the entity name it follows.

d) After the final entity name is a close parenthesis followed by a semicolon.

e) If there are any local ENTITY, SUBTYPE_CONSTRAINT, FUNCTION, RULE, PROCEDURE, or TYPE declarations, they are written next, indented two characters to the right of the left margin.

f) If there are any CONSTANT declarations, they are written next. The CONSTANT and END_CONSTANT keywords are written on separate lines, flush with the left margin. Each constant is written on a separate line indented two characters to the right of the left margin.

g) If there are any LOCAL declarations, they are written next. The LOCAL andEND_LOCAL keywords are written on separate lines, flush with the left margin. Each local is written on a separate line indented two characters to the right of the left margin.

h) The statements that make up the body of the RULE are written next, with each one on a separate line and indented two characters to the right of the left margin.

i) If there are any where rules, they come next. The WHERE keyword is written on a separate line, flush with the left margin. Each where is written on its own line, indented two characters to the right of the left margin.

j) Finally comes the END_RULE keyword on its own line, flush with the left margin, followed by a semicolon.

EXAMPLE

```
RULE rule_id FOR (entity_ref,
                  entity_ref);
  ENTITY rule_sub_ent;
    name : STRING;
  END_ENTITY;

  CONSTANT
    my_pi : REAL := 3.1415926;
  END_CONSTANT;
  LOCAL
    my_ent: rule_sub_ent;
  END_LOCAL;
  inst := rule_sub_ent('what');
WHERE
  WR1: expression;
  ...
END_RULE;
```

6.11 PROCEDURE

NOTE This clause reflects ISO/TC 184/SC 4 N2412, 6.1.10, “Algorithm layout”.

A procedure declaration is printed as follows:

a) It starts with the PROCEDURE keyword written on a new line flush with the left margin. Immediately following the PROCEDURE keyword is a space followed by the name of the procedure, followed by no space and an open parenthesis that marks the beginning of the formal parameters.

b) Each formal parameter is written on its own line, aligned with the one above it. The first formal parameter is written immediately after the open parenthesis with no space between it and the parenthesis.

EXAMPLE

```
PROCEDURE proc_id (formal_parameter;
                   formal_parameter);
  declarations
  CONSTANT
    my_pi : REAL := 3.1415926;
  END_CONSTANT;
  LOCAL
    my_inst : proc_sub_ent;
  END_LOCAL;
  my_inst := proc_sub_ent(23.0)
END_PROCEDURE;
```

7 Expressions

7.1 General

Expressions are the basic building blocks of the EXPRESS language. They are used to represent values, perform calculations, and define logical relationships.

NOTE ISO/TC 184/SC 4 N2412 does not provide guidance for how to layout the various expressions in EXPRESS. This clause establishes rules that are consistent with ISO/TC 184/SC 4 N2412 where possible.

The basic formatting rules for expressions are as follows:

a) If an expression fits on one line, it should be formatted that way.

b) If it does not fit on one line, it will be written to multiple lines with each line aligned with the expression’s start point on the first line and written with as many of the elements as possible on each line.

7.2 Unary MINUS expression

Write a hyphen ('-') followed by either a parenthetical expression, a literal value, or a qualifiable factor. There is no space between the hyphen and the following item.

EXAMPLE

```
-42
-(42 * 5 / 3)
```

7.3 NOT expression

Write the `NOT` keyword, followed by either a parenthetical expression, a literal value, or a qualifiable factor. There is no space between the keyword and the following item.

EXAMPLE

```
NOT(x + 5)
NOT y
```

7.4 Parenthetical expression

Write a left parenthesis, followed by the expression, followed by a right parenthesis. If the expression spans multiple lines, then each subsequent line is aligned so that its first non-white space character is in the column immediately following the left parenthesis that began the Parenthetical Expression.

EXAMPLE

```
(5+3)

(42 DIV 5 * 64 *
 23 + 15)
```

7.5 Unary PLUS expression

Write a plus sign ('+') followed by either a parenthetical expression, a literal value, or a qualifiable factor. There is no space between the plus sign and the following item.

EXAMPLE

```
+42
+(12 / 3)
```

7.6 Equal expression

Write an expression followed by an equal sign ('=') followed by an expression. There will be at least one space before and after the equal sign.

EXAMPLE

```
x = y + 15
```

7.7 Exponent expression

Write an expression followed by a double asterisk ('**') followed by an expression. There will be no space before or after the double asterisk symbol.

EXAMPLE

```
x**2
```

7.8 Greater than expression

Write an expression followed by a greater than symbol ('>') followed by an expression. There will be at least one space before and after the greater than symbol.

EXAMPLE

```
x > y
```

7.9 Greater than equal expression

Write an expression followed by a greater than or equal symbol ('>=') followed by an expression. There will be at least one space before and after the greater than or equal symbol.

EXAMPLE

```
x >= y
```

7.10 IN expression

Write an expression followed by the `IN` keyword followed by an expression. There will be at least one space before and after the `IN` keyword.

EXAMPLE

```
'this' IN ['some', 'that', 'this', 'other']
```

7.11 Instance equal expression

Write an expression followed by the instance equal symbol (':=:') followed by an expression. There will be at least one space before and after the instance equal symbol.

EXAMPLE

```
x :=: y
```

7.12 Less than expression

Write an expression followed by the less than symbol ('') followed by an expression. There will be at least one space before and after the less than symbol.

EXAMPLE

```
x < y
```

7.13 Less than or equal expression

Write an expression followed by the less than or equal symbol ('⇐') followed by an expression. There will be at least one space before and after the less than or equal symbol.

EXAMPLE

```
x <= y
```

7.14 LIKE expression

Write an expression followed by the `LIKE` keyword followed by an expression. There will be at least one space before and after the `LIKE` keyword.

EXAMPLE

```
x LIKE y
```

7.15 Not equal expression

Write an expression followed by the not equal symbol ('>') followed by an expression. There will be at least one space before and after the not equal symbol.

EXAMPLE

```
x <> y
```

7.16 Not instance equal expression

Write an expression followed by the not instance equal symbol (':>:') followed by an expression. There will be at least one space before and after the not instance equal symbol.

EXAMPLE

```
x :<>: y
```

7.17 Multiply expression

This expression object can have more than two elements.

Write each expression separated by the multiply symbol ('*'). There will be at least one space before and after the multiply symbol.

If all of the elements can't fit on one line then as many lines as necessary are used. Each line contains as many elements as possible with line breaks occurring just after the multiply symbol and the subsequent lines aligned with the first expression element on the line above.

EXAMPLE

```
x * y * z

x * y *
z * q
```

7.18 Addition expression

This expression object can have more than two elements.

Write each expression separated by the addition symbol ('+'). There will be at least one space before and after the addition symbol.

If all of the elements can't fit on one line then as many lines as necessary are used. Each line contains as many elements as possible with line breaks occurring just after the addition symbol and the subsequent lines aligned with the first expression element on the line above.

EXAMPLE

```
x + y + z

x + y +
z + q
```

7.19 Subtraction expression

This expression object can have more than two elements.

Write each expression separated by the subtraction symbol ('-'). There will be at least one space before and after the subtraction symbol.

If all of the elements can't fit on one line then as many lines as necessary are used. Each line contains as many elements as possible with line breaks occurring just after the subtraction symbol and the subsequent lines aligned with the first expression element on the line above.

EXAMPLE

```
x - y - z

x - y -
```

```
z - q
```

7.20 Division expression

This expression object can have more than two elements.

Write each expression separated by the division symbol ('/'). There will be at least one space before and after the division symbol.

If all of the elements can't fit on one line then as many lines as necessary are used. Each line contains as many elements as possible with line breaks occurring just after the division symbol and the subsequent lines aligned with the first expression element on the line above.

EXAMPLE

```
x / y / z

x / y /
z / q
```

7.21 AND expression

This expression object can have more than two elements.

Write each expression separated by the AND keyword. There will be at least one space before and after the AND keyword.

If all of the elements can't fit on one line then as many lines as necessary are used. Each line contains as many elements as possible with line breaks occurring just after the AND keyword and the subsequent lines aligned with the first expression element on the line above.

EXAMPLE

```
x AND y AND z

x AND y AND
z AND q
```

7.22 Compose expression

This expression object can have more than two elements.

Write each expression separated by the compose symbol ('||'). There will be at least one space before and after the compose symbol.

If all of the elements can't fit on one line then as many lines as necessary are used. Each line contains as many elements as possible with line breaks occurring just after the compose symbol and the subsequent lines aligned with the first expression element on the line above.

EXAMPLE

```
x || y || z

x || y ||
z || q
```

7.23 DIV expression

This expression object can have more than two elements.

Write each expression separated by the DIV keyword. There will be at least one space before and after the DIV keyword.

If all of the elements can't fit on one line then as many lines as necessary are used. Each line contains as many elements as possible with line breaks occurring just after the DIV keyword and the subsequent lines aligned with the first expression element on the line above.

EXAMPLE

```
x DIV y DIV z

x DIV y DIV
z DIV q
```

7.24 MOD expression

This expression object can have more than two elements.

Write each expression separated by the MOD keyword. There will be at least one space before and after the MOD keyword.

If all of the elements can't fit on one line then as many lines as necessary are used. Each line contains as many elements as possible with line breaks occurring just after the MOD keyword and the subsequent lines aligned with the first expression element on the line above.

EXAMPLE

```
x MOD y MOD z

x MOD y MOD
z MOD q
```

7.25 OR expression

This expression object can have more than two elements.

Write each expression separated by the OR keyword. There will be at least one space before and after the OR keyword.

If all of the elements can't fit on one line then as many lines as necessary are used. Each line contains as many elements as possible with line breaks occurring just after the OR keyword and the subsequent lines aligned with the first expression element on the line above.

EXAMPLE

```
x OR y OR z

x OR y OR
z OR q
```

7.26 XOR expression

This expression object can have more than two elements.

Write each expression separated by the XOR keyword. There will be at least one space before and after the XOR keyword.

If all of the elements can't fit on one line then as many lines as necessary are used. Each line contains as many elements as possible with line breaks occurring just after the XOR keyword and the subsequent lines aligned with the first expression element on the line above.

EXAMPLE

```
x XOR y XOR z

x XOR y XOR
z XOR q
```

7.27 Aggregate initializer expression

This is a sequence of expressions each separated by a comma and enclosed in square brackets.

Each expression is written with a comma separating any two adjacent expressions. At least one space is written after the comma with no space before the comma. The first expression starts just after the left square bracket and the right square bracket occurs just after the last expression.

If the aggregate initializer expression can't fit on one line then line breaks are placed just after the comma. Each element is written to a separate line aligned with the expression above it.

EXAMPLE

```
[42, x + y, 'this', .that.]

[15,
 a + very + long + expression,
 'this string',
 .enum_val.]
```

7.28 Call function expression

Write the name of the FUNCTION. If there are any parameters to the function, then the parameter expressions are enclosed in parentheses and separated by commas.

If there are parameters then the left parenthesis is written just after the function name with no space separating it from the function name. Each expression is written with a comma separating any two adjacent expressions. There will be at least one space after the comma and no space before the comma.

If more than one line is needed then the expressions are written one to a line with the line break occurring just after the comma. Each expression is aligned with the expression above it. The first expression starts just after the left parenthesis and the right parenthesis is written just after the final expression.

EXAMPLE

```
fun1(42, .t., 'this')

fun1(42,
     .t.,
     'this')
```

7.29 ENTITY constructor expression

Write the name of the ENTITY to be constructed.

If there are values needed to initialize any attributes, then the list of expressions is written just after the entity name and enclosed in parentheses and separated by commas.

This list begins with a left parenthesis just after the ENTITY name with no space between the parenthesis and the entity name. Each expression is written with a comma separating them. There is at least one space after the comma and no space before the comma. The right parenthesis follows immediately after the last expression.

If all of the expressions won't fit on one line then each expression is placed on its own line with line breaks occurring just after the comma. The first expression occurs just after the left parenthesis and on the same line. Each subsequent expression is aligned with the one above it.

EXAMPLE

```
entity1('this', 42, (1,2,4,6), .enum_val1.)

entity1('this',
        42,
        (1,2,4,6),
        .enum_val1.)
```

7.30 Enumeration reference expression

If the name of the TYPE declaration is provided, write the name of the TYPE declaration followed by a period ('.').

Write the enumeration value.

EXAMPLE

```
enum1

type1.enum2
```

7.31 Interval expression

Write an open curly brace {. Write the low value expression. Write the first operator. Write the item value expression. Write the second operator. Write the hi value expression. Write a close curly brace }.

If the expression can't fit on one line, then write the low value expression on the line with the left curly brace followed by the first operator. Write the item value expression on the next line followed by the second operator. The item value expression will be aligned with the low value expression on the line above. Finally, on the next line write the hi value expression followed by the right curly brace. The hi value expression will be aligned with the item value expression on the line above.

EXAMPLE

```
{1 < x < 10}

{1 <
 x <
 10}
```

7.32 Qualifiable factor expression

This qualifiable-factor object contains both the factor and the qualifiers.

Write the factor. For each qualifier, write the qualifier.

If they need more than one line, put as many qualified on each line as possible and align each subsequent line with the first qualifier on the line above.

EXAMPLE

```
var1\ENTITY1.attr1[23]

var1\ENTITY1
    .attr1[23]
```

7.33 QUERY expression

Write the `QUERY` keyword followed by an open parenthesis.

Next write the query variable id followed by the `<*` symbol.

Next write the source expression followed by the `|` symbol.

Next write the query expression followed by the close parenthesis.

If the query expression can't fit on one line then put line breaks just after the `<*` and `|` symbols. The source expression will be aligned with the query variable id. The query expression will be aligned with the source expression.

EXAMPLE

```
QUERY(var1 <* [1,2,3,4,5] | var1 < 3)

QUERY(var1 <*
      [1,2,3,4,5] |
```

```
        var1 < 3)
```

8 Statements

8.1 General

Statements are the executable parts of an EXPRESS schema. They are instructions that perform actions, control flow, and manipulate data. The statements are typically used within procedures, functions, and other control structures to define the behavior of the schema.

NOTE ISO/TC 184/SC 4 N2412 does not provide guidance for how to layout statements in EXPRESS. This clause establishes rules that are consistent with ISO/TC 184/SC 4 N2412 where possible.

The basic formatting rules for statements are as follows:

a) If a statement fits on one line, it should be formatted that way.

b) If it does not fit on one line, it will use as many lines as necessary and will be formatted appropriately for the specific statement.

8.2 ALIAS statement

Write the keyword `ALIAS` followed by the variable id followed by the keyword`FOR` followed by a general reference and set of qualifiers for the thing being aliased. Finish up with a ';' (semicolon) to end it.

EXAMPLE

```
ALIAS var_id FOR ent1\attr1;
```

8.3 Assignment statement

Write a pretty version of the general_ref followed by the assignment keyword ':=' followed by a pretty version of the expression followed by a ';' (semicolon). At least one space should be included before and after the assignment keyword ':='.

Multiple lines will be used depending on the expression itself.

EXAMPLE

```
i := 23 * 42 + x;
```

8.4 Call PROCEDURE statement

Write the name of the procedure.

If there are parameters to be passed to the procedure then write an open parenthesis followed by a list of the parameter expressions each separated by a comma. Finally write a close parenthesis.

Finish up with a semicolon.

If the expressions won't fit on a single line then place line breaks just after the comma. Each expression will be placed on its own line aligned with the expression on the line above. The first expression is placed on the line with the left parenthesis and just after the parenthesis.

EXAMPLE

```
proc_name(42,'this',.T.,.THAT.);

proc_name(42,
          'this',
          .T.,
          .THAT.);
```

8.5 CASE statement

Write the keyword CASE followed by the expression that is the followed by the keyword OF.

Write each of the case actions starting on a separate line indented two characters to the right from the left edge of the CASE keyword.

After the CASE Actions, if there is an otherwise clause, write the keywordOTHERWISE on a separate line, indented two characters to the right of the left edge of the CASE keyword. Follow the OTHERWISE keyword with a colon and the statement that is the otherwise clause. The colon should be right next to the OTHERWISE keyword with one or more spaces after the colon.

Finally, on a new line write the keyword END_CASE aligned with the CASE keyword. Follow the END_CASE keyword with a semicolon.

EXAMPLE

```
CASE <selector> OF
  label1: <stmt>
  label2, label3:
    <stmt>
  OTHERWISE: <stmt>
END_CASE;
```

8.5.1 CASE action

Each CASE Action begins on its own line, indented two characters to the right of the left edge of the CASE keyword.

Write each specified label separated by a comma.

Next write a colon next to the last label. The colon should have at least one space after it.

Next pretty print the statement that is the body. If the statement can't fit on the line with the label(s), then start it on the next line indented two characters to the right of the left edge of the first label.

EXAMPLE

```
label1, label2: <stmt>

label3, label4, label5:
  <stmt>
```

8.6 Compound statement

The compound statement begins with the BEGIN keyword and ends with the END keyword. Each of these keywords should be on its own line and aligned with each other. Following the END keyword there should be a semicolon.

Each of the enclosed statements should be written on a separate line indented two characters to the right of the left edge of the BEGIN keyword.

EXAMPLE

```
BEGIN
  a := 42 * i;
  RETURN(a);
END;
```

8.7 ESCAPE statement

The ESCAPE statement is self contained and consists of just the ESCAPE keyword followed by a semicolon.

EXAMPLE

```
ESCAPE;
```

8.8 IF statement

The IF statement consists of the keyword IF followed by the expression that is the condition. Next follows the THEN keyword and the statements that constitute the then portion. If there is an else part of the if then theELSE keyword is written followed by the else statements. The THEN andELSE keywords each have their own line and are aligned with the IF keyword. The THEN and ELSE statements are indented two characters to the right of the left edge of the associated keyword.

Finally, on a new line, aligned with the IF keyword we write the END_IF keyword followed by a semicolon.

EXAMPLE

```
IF <condition>
THEN
  a := 42 * i;
  RETURN(a);
ELSE
  a := x / 15;
  RETURN(a);
END_IF;
```

8.9 NULL statement

The NULL statement is self contained and consists of only a semicolon.

EXAMPLE

```
;
```

8.10 REPEAT statement

The REPEAT Statement consists of the REPEAT keyword followed by the Increment Controls, followed by any While Controls, followed by any Until Controls, followed by a semicolon, followed by the statements that should be executed during each loop. Finally, there is a END_REPEAT keyword written on its own line aligned with the REPEAT keyword and followed by a semicolon.

The Increment Controls, While Controls, and Until Controls should be indented four characters to the right of the left edge of the REPEAT keyword.

NOTE The SC4 Supplementary Directives require the Increment Controls to be on a separate line after the REPEAT keyword but SC4 practice is to include the Increment Controls on the same line as the REPEAT keyword and separated from the REPEAT keyword by a single space.

EXAMPLE

```
REPEAT i := 0 TO 15 BY 1
    WHILE x > 20
    UNTIL y < 50;
  a := 42 * x / y;
  b := x * i DIV y;
END_REPEAT;
```

8.10.1 Increment control

An Increment Control consists of a variable id followed by an assignment operator followed by an expression that calculates the initial value of the variable followed by the TO keyword, followed by an expression that calculates the end value of the variable, optionally, followed by the BY keyword and an expression the calculates the increment value.

EXAMPLE

```
a := 1 TO 5
a := 1 TO 5 BY 0.5
```

8.10.2 While control

A While Control consists of the keyword `WHILE` followed by an expression that returns false when the `REPEAT` should stop.

EXAMPLE

```
WHILE a < 5
```

8.10.3 Until control

An Until Control consists of the keyword `UNTIL` followed by an expression that returns true when the `REPEAT` should stop.

EXAMPLE

```
UNTIL x > 50
```

8.11 RETURN statement

The RETURN statement is fairly simple and consists of the keyword `RETURN` optionally followed by an open parenthesis, an expression which calculates the value to be returned, a close parenthesis.

Finally, it is terminated with a semicolon.

EXAMPLE

```
RETURN;
RETURN(42);
```

8.12 SKIP statement

The SKIP statement is self contained and consists of the keyword `SKIP` followed by a semicolon.

EXAMPLE

```
SKIP;
```

Bibliography

[1] EE-Pretty-Print, *Express Engine Pretty Printer Documentation*, Craig Lanning, Thomas Thurman.

[2] EXPRESS-Engine, *Express Engine*, Craig Lanning, Thomas Thurman, et al.

Price based on 22 pages